

Published on Linux Journal (<http://www.linuxjournal.com>)

The Linux Kernel Cryptographic API

By James Morris

Created 2003-04-01 02:00

This article provides a brief overview of the new cryptographic API for the Linux kernel. It is aimed at anyone with a technical interest in Linux, such as system administrators, and other curious people who would like to gain insight into the API's design, implementation and application. Some knowledge of kernel internals is useful but not essential for a broad understanding of API concepts.

The history of this API is short. Not long before the Halloween 2002 kernel feature-freeze, an IPSec implementation being developed by Dave Miller and Alexey Kuznetsov became slated for inclusion into the 2.6 kernel. IPSec requires cryptographic support within the kernel, which along with an increasing general need for kernel cryptography, prompted the development of a new cryptographic API.

Design

Although initially aimed at supporting IPSec, the API has been designed as a general-purpose facility, with potential applications including encrypted files, encrypted filesystems, strong filesystem integrity, the random character device (/dev/random), network filesystem security (for example, CIFS) and other kernel networking services requiring cryptography.

A specific design requirement was that the API work directly in place on page vectors. A page is the primary unit of memory managed by the kernel. A page vector-based API allows for deep integration with kernel substructures, such as the VFS and networking stack, as well for as scatter-gather operations. In the case of IPSec, cryptographic transforms may be applied directly to discontiguous memory pages associated with network packets.

Simplicity was a significant design goal, which is always a good idea in general, and particularly important for kernel and security code.

Deployment flexibility was another goal. For example, the API has a flexible policy toward algorithms; they can be loaded dynamically as kernel modules, without the API needing to know anything about them in advance.

Future design goals include:

- Hardware support for cryptographic accelerator cards and NICs with IPSec offload.
- Support for specification of algorithm preferences when multiple implementations are available, for example, optimized assembler versions and various hardware implementations.
- Asymmetric cryptography support (RSA), which may be needed in the kernel to support multicast IPSec and kernel module signature verification. This may be a contentious issue, as asymmetric cryptography is generally slow and complicated—both are very good reasons to exclude it from the kernel.
- A unified API for user-space applications wishing to utilize available cryptographic hardware, such as SSL, IPSec key exchange, secure routing protocols and DNSSec.

- Further optimizing the API memory footprint to cater to embedded systems scenarios.

Algorithms

Three types of algorithms are currently supported by the API:

1) Digests (one-way hash functions)—these take arbitrary messages and produce short, fixed-length message digests. To be one-way, the hash function must be designed so it is easy to generate the hash but difficult to compute the original message from the hash. For cryptographic purposes, hash functions need to be collision-resistant, so that it is difficult for two messages to hash to the same value. Applications include ensuring data integrity and generating message authentication codes for network protocols. Examples of digest algorithms are MD5 and SHA1.

A message authentication scheme called HMAC (RFC2104) is included within the API, which will operate on any standard digest algorithm. This is currently used to generate authentication data for IPSec packets.

2) Ciphers—these algorithms implement symmetric key encryption, where a plain-text message is encrypted with a key to produce ciphertext. Generally, the same key is used to decrypt the ciphertext back into the original plain text. It should be easy to encrypt and decrypt messages with the key (which must be kept secret) but difficult to do so without it. Applications include encrypting data to ensure privacy and generation of message authentication codes. Examples of cipher algorithms are Triple DES, Blowfish and AES.

There are two types of ciphers: block ciphers operate on fixed-length blocks of data (e.g., 16 bytes), and stream ciphers use a key stream to operate on as little as one bit of data at a time.

Ciphers also may operate in a variety of modes, such as Electronic Codebook (ECB), where each block of plain text is simply encrypted with the key, and Cipher Block Chaining (CBC), where the previously encrypted block is fed into the encryption of the next block.

3) Compression—this is often used in conjunction with encryption so that it is more difficult to exploit weaknesses related to the original plain text as well as to speed up encryption (i.e., compressed plain text is shorter). By definition, encrypted data should be difficult to compress, but this adversely affects performance over links that normally utilize compression. Compressing data before encryption helps reduce this performance hit in many cases. Examples of compression algorithms are LZS and Deflate.

So far, algorithm implementations from well-known sources have been adapted for use with the API, as they are more likely to have been reviewed and widely tested. For inclusion into the mainline kernel, algorithms generally must be patent-free (e.g., IDEA will not be a candidate for inclusion until around 2011), based on open, recognized standards and submitted with a set of test vectors.

Page Vectors

Before discussing the API structure, let's briefly look at memory pages and page vectors. As mentioned previously, a page is the fundamental unit of memory managed by the kernel (on i386, pages are 4KB in size). Consider a buffer containing, say, 1,460 bytes of user-space data. It belongs to a specific page in the kernel, offset from the start of the page by some amount, and has a length of 1,460 bytes. This buffer can be represented as a page-based tuple:

```
{ page, offset, length }
```

An interface, such as the cryptographic API that works directly with pages, needs to deal with this tuple, or page vector. An existing kernel data structure called a scatterlist is employed, which contains a page vector and normally is used for scatter-gather DMA operations.

The cryptographic API uses scatterlists to operate on arrays of discontinuous page vectors. The primary purpose of scatter-gather in the kernel is to avoid unnecessary copying of data. It also seems to result in cleaner code. Many readers will be familiar with scatter-gather I/O in the form of the `readv()` and `writew()` system calls. The kernel cryptographic API uses the same general concept but operates on pages instead of plain memory buffers.

API Structure

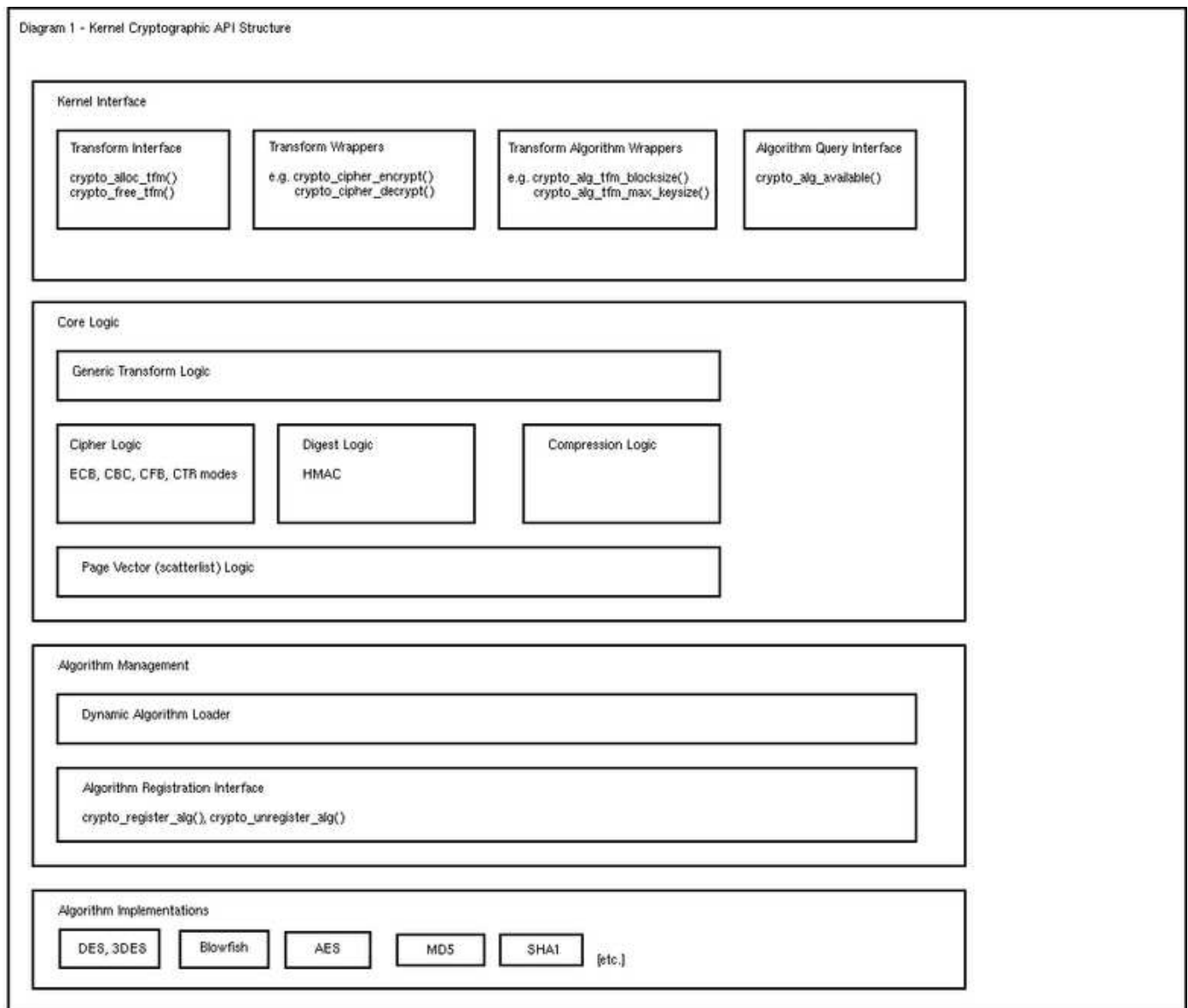
The API deals with two primary objects:

- Algorithm implementations—kernel modules that contain the underlying algorithm code.
- Transforms—objects that instantiate algorithms, manage internal state and handle common implementation logic. Transforms are managed by `crypto_alloc_tfm()` and `crypto_free_tfm()`. A set of API wrappers are provided to simplify transform use and to allow the properties of a transform's underlying algorithm to be queried.

The following pseudo-code demonstrates a typical use of the transform interface, where some kernel code needs to encrypt data using the Blowfish cipher in electronic codebook (ECB) mode:

```
tfm = crypto_alloc_tfm("blowfish",  
                      CRYPTO_TFM_MODE_ECB);  
crypto_cipher_setkey(tfm, key, keylength);  
crypto_cipher_encrypt(tfm, &scatterlist,  
                     numlists);  
crypto_free_tfm(tfm);
```

As shown in Figure 1, the API is layered so that core logic is hidden from cryptography users and algorithm implementors. This core logic includes generic transform management, scatterlist manipulation and abstraction of underlying algorithms. Further down, per-algorithm-type logic is handled, such as cipher processing modes and utilizing digests for generating message authentication codes.



[1]

Figure 1. Structure of the Kernel Cryptographic API

The algorithm management layer contains logic for locating, loading and reference counting algorithm implementations. The latter is required to prevent nasty things from happening if an attempt is made to unload an algorithm module that is still in use.

An algorithm runtime query interface is provided so that calling code can determine which algorithms are available on the system. This is primarily intended for use by key negotiation protocols, such as ISAKMP/IKE.

Finally, the algorithm registration interface allows modules to register one or more algorithms, specifying various properties such as the name of the algorithm, its block size and minimum and maximum key sizes. The list of currently registered algorithms and their properties may be viewed in `/proc/crypto`.

Conclusions

This is still a young API that is likely to evolve somewhat, especially if some of the future design goals listed here are implemented.

In terms of API users, IPSec works and performs well, especially for a first cut with no performance optimizations. Existing kernel components that need cryptography are expected to convert to the new API over time, and hopefully, cool new projects will be developed because of it.

Acknowledgements

Many thanks to David Miller and Nancy Chan for reviewing this article.

Resources [2]

email: jmorris@intercode.com.au [3]

James Morris is a software developer involved with the Netfilter, LSM, SELinux and Linux kernel cryptographic API projects. He works as an independent consultant in Sydney, Australia.

Source URL: <http://www.linuxjournal.com/article/6451>

Links:

[1] <http://www.linuxjournal.com/files/linuxjournal.com/linuxjournal/articles/064/6451/6451f1.png>

[2] <http://www.linuxjournal.com/files/linuxjournal.com/linuxjournal/articles/064/6451/6451s1.html>

[3] <mailto:jmorris@intercode.com.au>